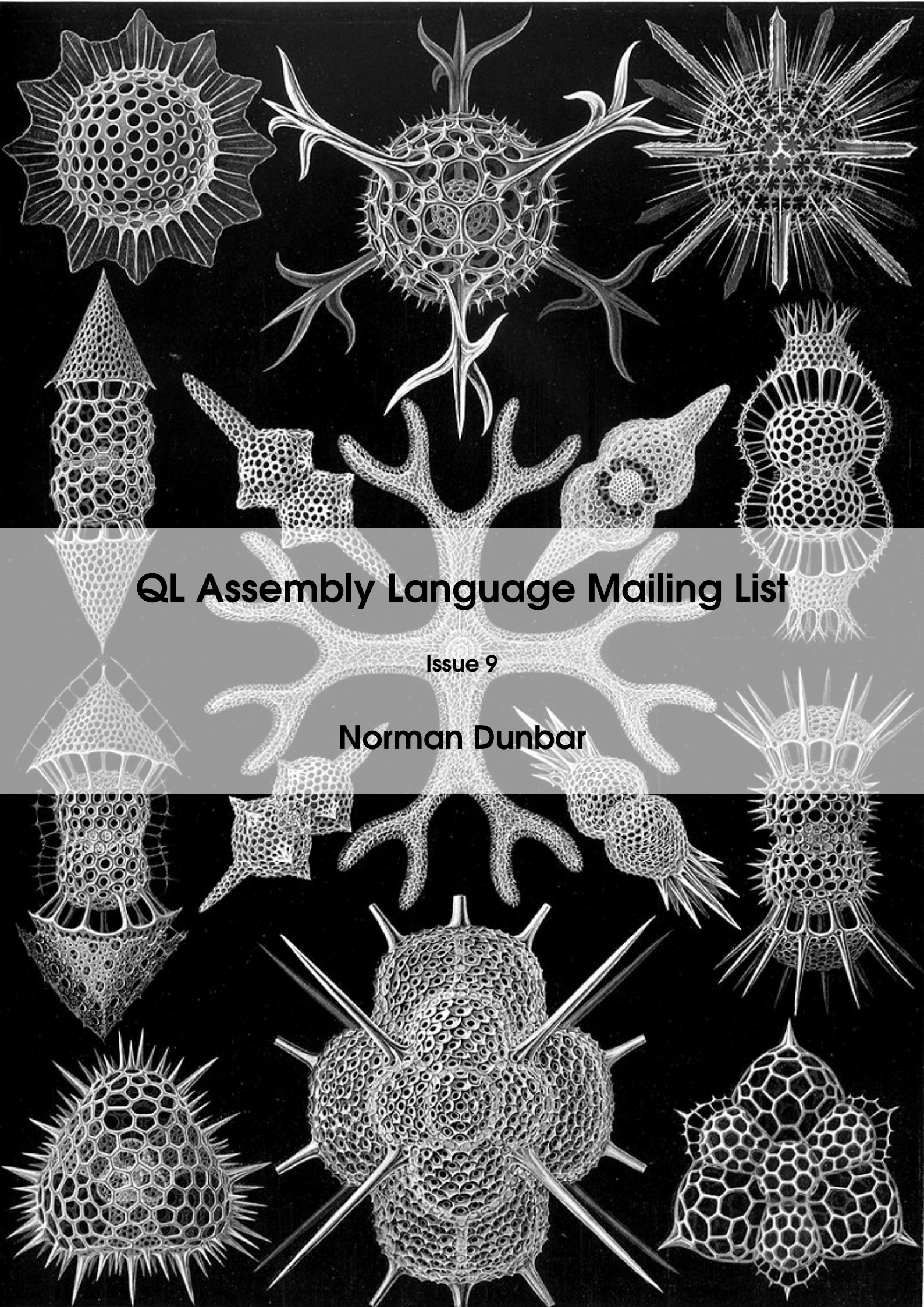


QL Assembly Language Mailing List

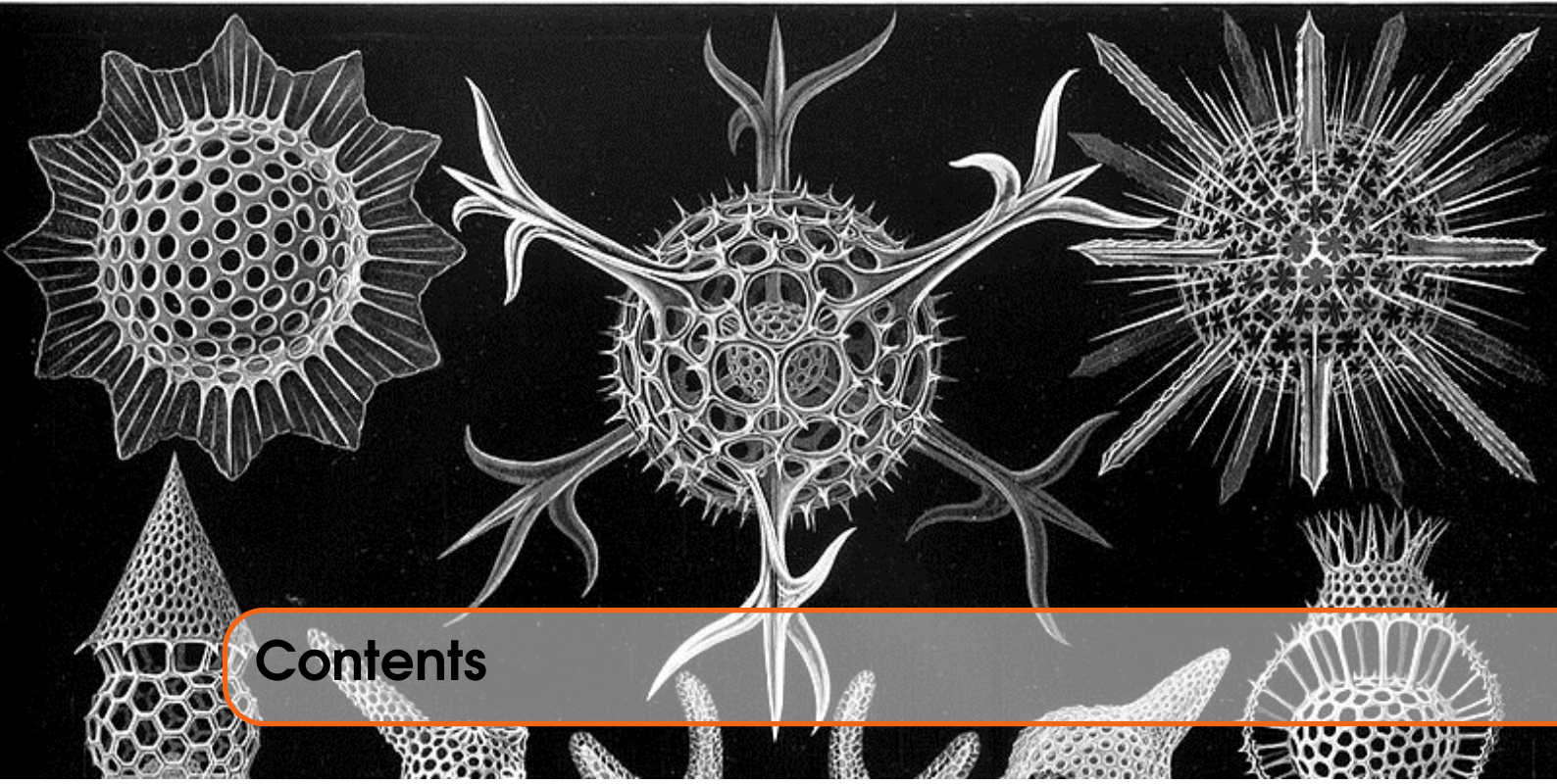
Issue 9

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

Download from:
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/releases/tag/Issue_9

This pdf document was created on *14/11/2021* at *16:57:24*.



Contents

1 Preface 7

1.1 Feedback 7

1.2 Subscribing to The Mailing List 7

1.3 Contacting The Mailing List 8

2 News 9

3 Feedback on Issue 8 11

4 QL2WIN 13

4.1 The Code 13

4.2 Filter Chains 18

5 Win2QL 21

5.1 Changes From Ql2win 21

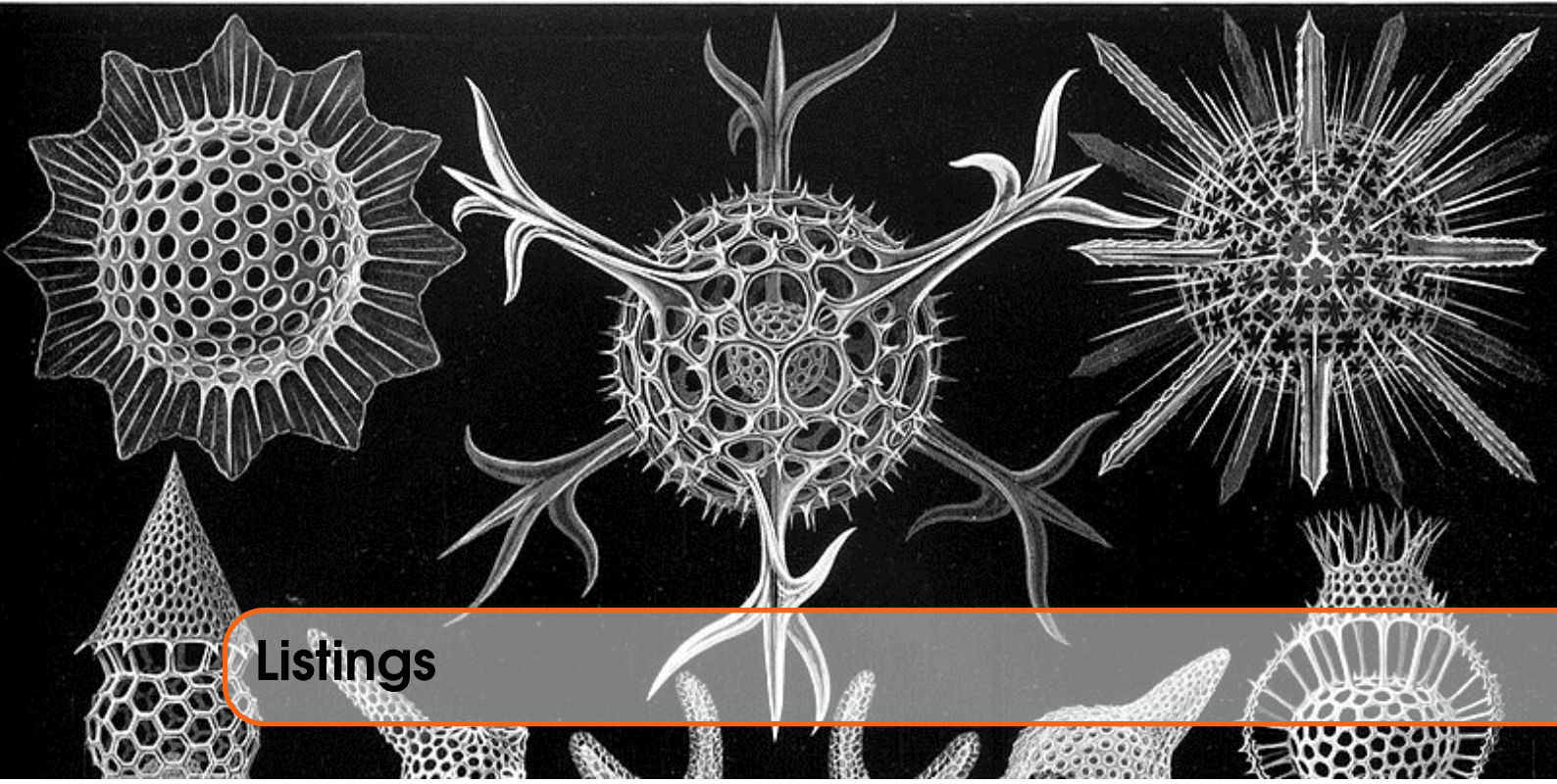
6 Circular Buffers 23

6.1 How Big is my Buffer? 24

6.2 Buffer Structure 24

6.3 The Buffer Handling Code 25
6.3.1 Allocate a New Buffer . . . 25
6.3.2 Buffer Size Adjustments . . . 27
6.3.3 Free a Buffer . . . 28
6.3.4 Buffer Check . . . 29
6.3.5 Write Data to a Buffer . . . 30
6.3.6 Read Data from a Buffer . . . 31
6.3.7 Is the Buffer Full? . . . 33
6.3.8 Is the Buffer Empty? . . . 34
6.3.9 How Much Space is Used? . . . 35
6.3.10 How Much Space is Free? . . . 36
6.3.11 Flushing Buffers . . . 37
6.3.12 Incrementing Head and Tail Offsets . . . 38
6.3.13 QPC2 Bug? My Mistake? . . . 40
6.4 Test Harness 40

7 Image Credits . . . 43



Listings

3.1 Rnd 1 to 6 function . . . 11
4.1 Ql2win: Equates . . . 14
4.2 Ql2win: Job Header . . . 14
4.3 Ql2win: Parameter checks . . . 15
4.4 Ql2win: Constants . . . 15
4.5 Ql2win: readLoop . . . 16
4.6 Ql2win: gotLine . . . 16
4.7 Ql2win: Adding a CR . . . 17
4.8 Ql2win: putLine . . . 17
4.9 Ql2win: Exit . . . 18
4.10 Ql2win: Buffer . . . 18
5.1 Win2ql: Comments . . . 21
5.2 Win2ql: Job Header . . . 22
5.3 Win2ql: Removing a CR . . . 22
6.1 Allocating a circular buffer . . . 25
6.2 Adjusting a buffer's size . . . 27
6.3 Freeing a circular buffer . . . 28
6.4 Validating a buffer address . . . 29
6.5 Writing data to a buffer . . . 30
6.6 Reading data from a buffer . . . 31

6.7 Is buffer full . . . 33
6.8 Is buffer empty . . . 34
6.9 Buffer space used . . . 35
6.10 Buffer free space . . . 37
6.11 Flushing buffers . . . 37
6.12 Incrementing an offset using AND . . . 38
6.13 Incrementing an offset using DIVU . . . 39
6.14 Correct offset incrementing with DIVU . . . 39
6.15 Incorrect offset incrementing with DIVU . . . 39
6.16 Test harness for cBuffer code . . . 41

1. Preface

1.1 Feedback

Please send all feedback to assembly@qdosmsq.dunbar-it.co.uk. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LaTeXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to http://qdosmsq.dunbar-it.co.uk/mailinglist and clicking on the link "Subscribe to our Newsletters".

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

assembly@qdosmsq.dunbar-it.co.uk

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.

2. News

The news this time is that there is an actual printed copy of the *QL Assembly Language* book available. This has been created, with my blessings, by a QL Forum member named “Tinyfpga” who prefers to read paper books as opposed to PDF files. I can’t disagree!

The original, and always the latest PDF version (I keep fixing small errors, grammar, spelling etc) will always be found at:

https://github.com/NormanDunbar/QLAssemblyLanguageBook/releases/latest.

If you prefer the paper version, it’s a “print on demand” thing. “Tinyfpga” has posted these instructions on the QL Forum:

https://qlforum.co.uk/viewtopic.php?f=9&t=3294&start=40#p35207

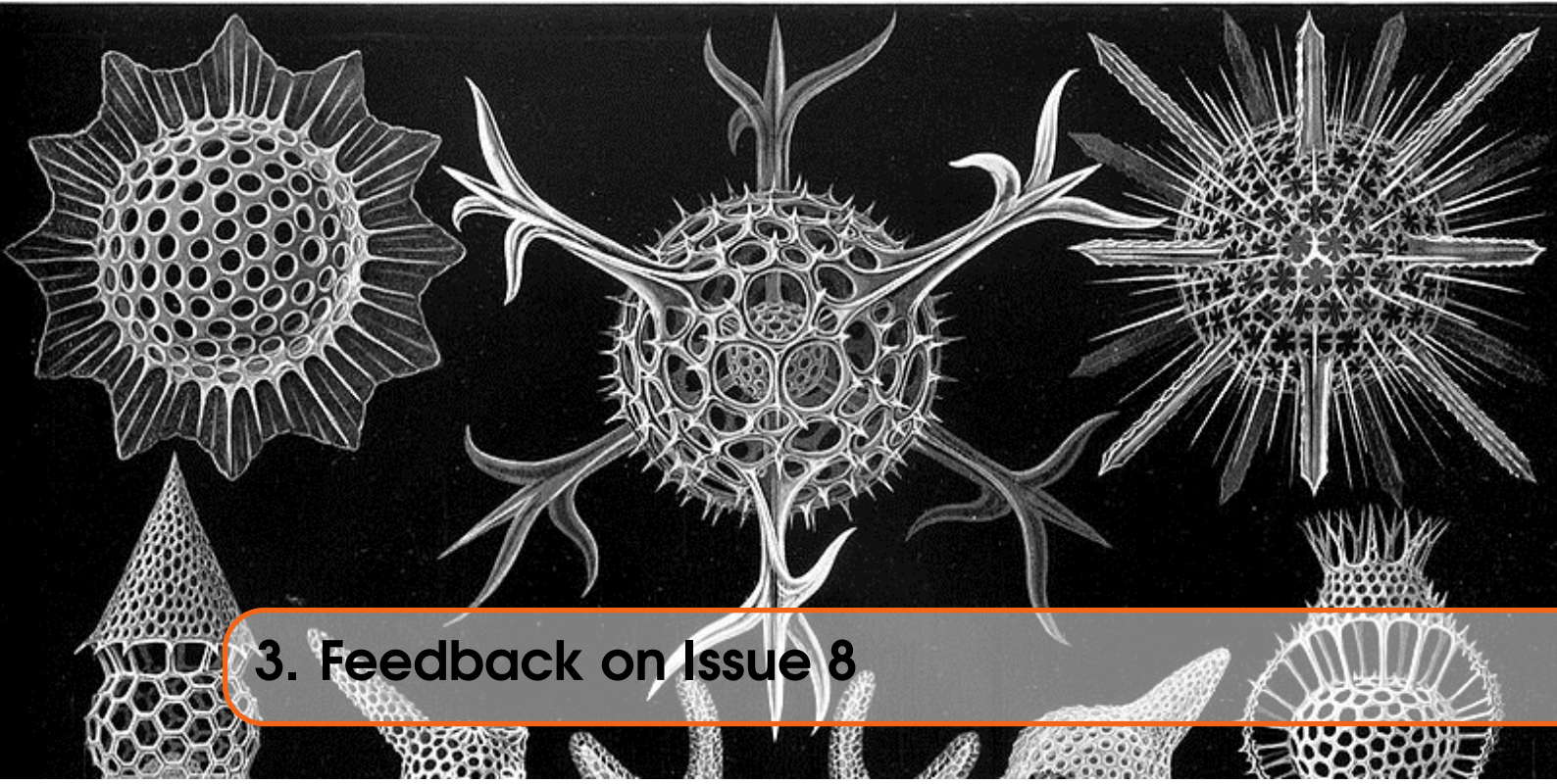
I’ve modified them slightly as it looks like the process has changed a little since first being published.

As mentioned in an earlier post I decided to "publish" (with permission) Norman Dunbar’s PDF book titled ’QL Today’s QL Assembly Language Programming Series - Book one’ as a real book.

I decided to publish it as an A4 354 page, lay-flat book for easy referencing. The book costs £7.04 plus £3.50 for postage and packaging direct from the printers. The method I have used for very low cost one-off printing to order means that anyone wanting to buy a book has to use my account to do so, as follows:

- Go to www.bookprintinguk.com
- Login as documentsforsms@gmail.com Password - forsms11 (That’s two digit ones on the end)
- Click on the Image of the book
- When the “What would you like to do” page comes up, click on the “Order More” link
- Select "order one copy" and then go next etc until you get to buy with your own name and address.

Hopefully, the process still remains the same.



3. Feedback on Issue 8

No long after release, I spotted a bug in the randomisation chapter in Issue 8. On page 49, in *Listing 7.3: Rnd 1 to 6 function - Part 1*, I had a comment on line 67 which mentioned "divide by 65536". That was complete nonsense, as that would have cleared the high word and not the low word.

Marcel spotted the error and advised me that the code was clearing overflow, not dividing. My mistake.

Marcel also spotted something else in the code I had blatantly stolen from the SMSQ sources.

It appears that the code in the rnd routine where we have this extract:

```
rnd
    ...
    mulu    #$c12d,d0           ; HHHH * 49453
    mulu    #$712d,d4           ; LLLL * 28973
  ...
    clr.w   d0                  ; HHHH 0000 (Divide by 65536)
  ...
```

Listing 3.1: Rnd 1 to 6 function

could possibly be a typo! He believes that the code should be calculating a 32 bit by 16 bit multiplication – the expression:

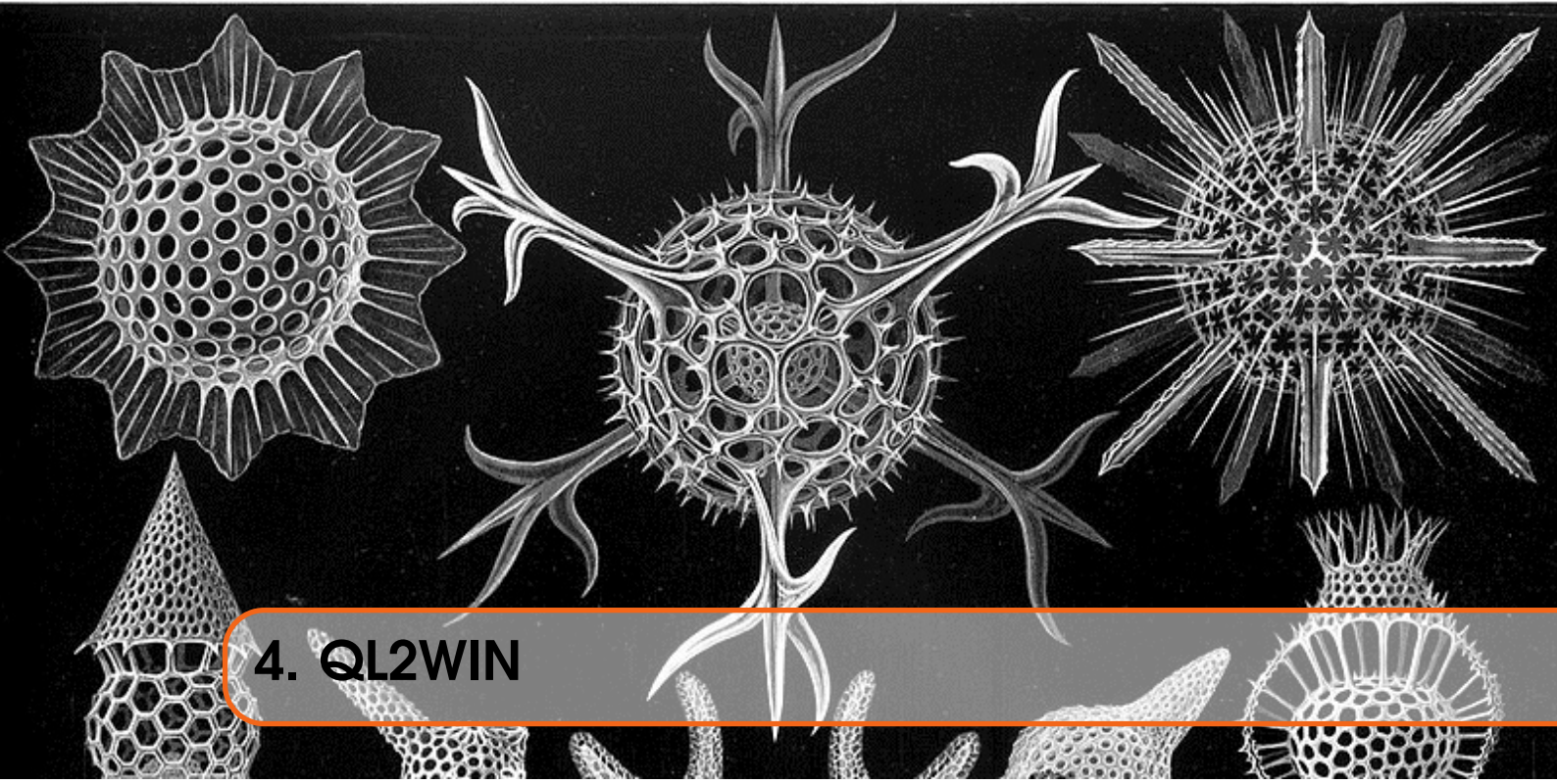
$$myRandSeed = myRandSeed * \$712D + 1$$

However, it seems to be this instead:

$$myRandSeed = myRandSeed * \$712D + ((myRandSeed \& \$F0000) * \$5000) + 1$$

Why both halves of the 32 bit random seed are not being multiplied by $712d is unknown, perhaps the used of $c12d is a typo, perhaps it's deliberate to make it more random. Nobody knows!

Thanks Marcel.



4. QL2WIN

From time to time I have to use Windows, or at least, attempt to open a file created on a Windows box while using my QL[1]. Usually, I open the file in a text editor of some kind, change the line endings setting and save the file that way, or I can use a myriad of Linux utilities to do the conversion. There are quite a few. However, this wouldn't be an ePeriodical on the use of QL Assembly Language if I didn't do it on a QL!

Given the above, I present for your wonderment and amazement, a small utility to convert a QL file to Windows format. Yes! I know! I said that I had occasion to open a Windows file on my QL, but check out the next chapter.....

4.1 The Code

It has been at least one issue, also known as "over a year", since I last wrote a YAF[2] utility. If you have missed them, then this is indeed a YAF. To convert a file from QL format with `CHR$(10)` (linefeed) line endings to Windows `CHR$(13)/CHR$(10)` (carriage return/linefeed) line endings, you simply have to:

```
EX ram1_ql2win_bin, ram1_ql_text_file, ram1_windows_txt_file
```

Listing 4.1 is the start of the code and covers a few equates and such like that I will be using through the code. As with many of my YAFs, there are only two channels required to be passed; the input QL file and the output Windows file. As I will not be faffing around in subroutines – given the extreme briefness of the code – the input channel id will be on the stack at 2(A7) while the output channel id ill be on the stack at 6(A7). The word at the top of the stack will hopefully be 2 for the number of opened channels passed.

[1] Don't ask!
[2] Yet Another Filter

```
;-------------------------------------------------------------
; QL2WIN:
;
; This filter converts QL or Linux line endings to Windows
; format.
;
; EX ql2win_bin, input_file, output_file_or_channel
;-------------------------------------------------------------
; 21/02/2021 NDunbar Created for QDOSMSQ Assembly Mailing List
;-------------------------------------------------------------
; (c) Norman Dunbar, 2021. Permission granted for unlimited use
; or abuse, without attribution being required. Just enjoy!
;-------------------------------------------------------------


;-------------------------------------------------------------
; How many channels do I want?
;-------------------------------------------------------------
numchans    equ     2           ; How many channels required?

;-------------------------------------------------------------
; Stack stuff.
;-------------------------------------------------------------
sourceId    equ     $02         ; Offset(A7) to input file id
destId      equ     $06         ; Offset(A7) to output file id

;-------------------------------------------------------------
; Other Variables
;-------------------------------------------------------------
err_bp      equ     -15
err_eof     equ     -10
me          equ     -1
timeout     equ     -1
lf          equ     $0a
cr          equ     $0d
```

Listing 4.1: Ql2win: Equates

Following on, we have Listing 4.2 which is the standard QDOSMSQ job header. There's nothing much of interest to see here, and further discussion would be fruitless. Lets move on!

```
;=============================================================
; Here begins the code.
;-------------------------------------------------------------
; Stack on entry:
;
; $06(a7) = Output file channel id.
; $02(a7) = Source file channel id.
; $00(a7) = How many channels? Should be $02.
;=============================================================
start
    bra.s   checkStack
    dc.l    $00
    dc.w    $4afb
name
    dc.w    name_end-name-2
    dc.b    'QL2WIN'
name_end    equ     *
```

```
52
53 version
54     dc.w     vers_end-version-2
55     dc.b     'Version 1.00'
56 vers_end     equ        *
```

Listing 4.2: Ql2win: Job Header

Listing 4.3 is where we check the parameters passed on the stack. We should have been passed 2 channels and a word informing us of same. The code checks that all is well, and if not, we exit with a bad parameter error.

```
57 ;-------------------------------------------------------------------
58 ; Check the stack on entry. We only require NUMCHAN channels.
59 ; Anything other than NUMCHANS will result in a BAD PARAMETER
60 ; error on exit from EW (but not from EX).
61 ;-------------------------------------------------------------------
62 checkStack
63     cmpi.w   #numchans,(a7)     ; Two channels is a must
64     beq.s    ql2win             ; Ok, skip error bit
65
66 bad_parameter
67     moveq    #err_bp,d0         ; Guess!
68     bra      errorExit          ; Die horribly
```

Listing 4.3: Ql2win: Parameter checks

Continuing on from Listing 4.3, we have a number of constants. These are values that will be needed at various places in the code, but which are stored in spare registers to speed up the code by not having to worry about getting stuff out of buffers; or from the stack; and such like. Listing 4.4, which follows, shows that the utility is written to hold the read and write timeout in register `D3`, the read/write buffer size in `D4`, the buffer address which is used for reading and writing is held in `A3` while `A4` and `A5` hold the channel ids for the source and destination channels respectively.

```
69 ;-------------------------------------------------------------------
70 ; Initialise a couple of registers that will keep their values
71 ; all through the rest of the code. These are:
72 ;
73 ; D3 holds the read and write timeout value, -1.
74 ; D4 holds the buffer size for reading into, buffSize.
75 ; A3 holds the buffer for reading and writing.
76 ; A4 holds the source channel id.
77 ; A5 holds the destination channel id.
78 ;-------------------------------------------------------------------
79 ql2win
80     moveq    #timeout,d3        ; Timeout
81     moveq    #buffSize,d4       ; Storage for buffer size for D2
82     lea      buffer,a3          ; Start of (write) buffer
83     move.l   sourceID(a7),a4    ; Source channel id
84     move.l   destId(a7),a5      ; Destination channel id
```

Listing 4.4: Ql2win: Constants

These constants will be swapped into the registers that need them as the code progresses. Why bother with this? Well register to register access is much faster than memory to register access, and while it might not speed things up for the sizes of the files I use, on the odd occasions, it might be useful in bigger programs where there needs to be a lot of this sort of thing.

```
85  ;-------------------------------------------------------------------
86  ; The main loop starts here. Read a single byte, check for EOF.
87  ;
88  ; D0 = 2 (io_fline)          Error code
89  ; D1.W                       Bytes read into buffer
90  ; D2.W = Buffer Size         Preserved
91  ; D3.W = timeout.            Preserved
92  ; A0.L = Channel ID.         Preserved
93  ; A1.L = Start of buffer.    Updated buffer (A1 + D2.W)
94  ;-------------------------------------------------------------------
95  readLoop
96      moveq    #io_fline,d0       ; Fetch lines ending with LF
97      move.w   d4,d2              ; Buffer size
98      movea.l  a4,a0              ; Channel to read
99      movea.l  a3,a1              ; Read buffer start
100     trap     #3                 ; Read a line from input file
101     tst.l    d0                 ; OK?
102     beq.s    gotLine            ; Yes
103     cmpi.l   #ERR_EOF,d0        ; All done yet?
104     beq      allDone            ; Yes.
105     bra      errorExit          ; Oops!
```

Listing 4.5: Ql2win: readLoop

The top of the main loop for the utility is shown in Listing 4.5. Here we see the use of the `io_fline` function to read a string of bytes, from a channel, into a buffer. The string of bytes is terminated by a linefeed character, and the maximum number of bytes to be read is determined by the value in `D2.W`.

Don't do as I did, and use `io_sstrg` instead. Because that one fills the buffer regardless of where it finds a linefeed in the bytes being read. I spent ages looking for a bug in my code and had my QDOS Companion open at the `io_sstrg` page instead of `io_fline`. Sigh!

The code in Listing 4.5 sets up the registers to read from the source file and reads it. If the read was successful, we skip to the code in Listing 4.6 to process the bytes just read, otherwise we have to check for End Of File. If we find EOF, we can bale out and close the file on the way, otherwise we have an error and exit the utility via the error handling code.

```
106 ;-------------------------------------------------------------------
107 ; At this point, we have a string and a clean read with no
108 ; errors. Check if we have read an entire line before we try to
109 ; convert this to Windows format.
110 ;
111 ; D1.W = bytes read into buffer, inc LF.
112 ; A1.L = one past where the LF should be.
113 ; If -1(a1) == lf we have a whole string.
114 ; Else write out what we have and read more of the same string.
115 ;-------------------------------------------------------------------
116 gotLine
117     move.w   d1,d2              ; Bytes read, required for write
118     cmpi.b   #lf,-1(a1)         ; Did we read the whole line?
119     bne.s    putLine            ; No, write out what we got
```

Listing 4.6: Ql2win: gotLine

The number of bytes read into the buffer is copied from `D1.W` to `D2.W` as we need `D2.W` to be correctly set for writing the bytes back out to the destination channel.

A1.L has been adjusted to point at the character above the trailing linefeed, if there is one, so we can check the character previous to that one. If that character is not a linefeed, then our buffer is too small to be able to read the entire line from the input channel. In this case, we simply skip to Listing 4.8 where we will write the data we have in the buffer, unchanged, to the destination channel.

If we have found a linefeed character, we drop into Listing 4.7 to process the line further, if necessary.

```
120 ;-------------------------------------------------------------------
121 ; We have read at least the end of a line and have the LF at
122 ; the correct place in the buffer. If the character before it
123 ; is a CR ignore it and write out, otherwise insert a CR before
124 ; the LF and write it all out.
125 ;-------------------------------------------------------------------
126     cmpi.b  #cr,-2(a1)          ; Already Windows format?
127     beq.s   putLine             ; Yes, ignore CR and write out
128     move.b  #cr,-1(a1)          ; Insert CR
129     move.b  #lf,(a1)            ; Needs LF also
130     addq.w  #1,d2               ; Update count for the CR
```

Listing 4.7: Ql2win: Adding a CR

It is possible that the file we are reading is already in Windows format. Before we go ahead and write a carriage return character just before the linefeed, we better check! If the character is a carriage return, we jump off to Listing 4.8 to write the line out unchanged.

Assuming the file is not already in Windows format, we replace the linefeed at -1(A1) with a carriage return and then add in a new linefeed at (A1). This is why we made the buffer big enough for an extra 2 characters, to give us room to add in the required carriage return.

As we have added in an additional character to the buffer, we need to update D2.W which currently holds the number of bytes we read in. This is used to determine how many bytes will be written to the destination file, which coincidentally enough, happens to be where we drop in next; to Listing 4.8.

```
131 ;-------------------------------------------------------------------
132 ; Write out the contents of the buffer.
133 ;
134 ; D0 = 7 (io_sstrg)          Error code
135 ; D1.W                       Bytes written to channel
136 ; D2.W = Buffer Size         Preserved
137 ; D3.W = timeout.            Preserved
138 ; A0.L = Channel ID.         Preserved
139 ; A1.L = Start of buffer.    Updated buffer (A1 + D2.W)
140 ;-------------------------------------------------------------------
141 putLine
142     moveq   #io_sstrg,d0        ; Send strings
143     movea.l  a5,a0              ; Dest channel id
144     movea.l  a3,a1              ; Write buffer
145     trap    #3                  ; Do it
146     tst.l   d0                  ; OK?
147     beq.s   readLoop            ; Yes, keep going
148     bra.s   errorExit           ; No.
```

Listing 4.8: Ql2win: putLine

The code at putLine uses io_sstrg to write data from a buffer to a channel. The number of bytes is determined by D2.W. The required registers are set up by copying in those required from our

constants where they have been sitting, waiting their turn of action! The remainder of the code, as seen in Listing 4.9, handles errors and exiting from the utility when all is done.

```
149 ;---------------------------------------------------------------
150 ; No errors, exit quietly back to SuperBASIC.
151 ;---------------------------------------------------------------
152 allDone
153     moveq   #0,d0
154
155 ;---------------------------------------------------------------
156 ; We have hit an error so we copy the code to D3 then exit via
157 ; a forced removal of this job. EXEC_W/EW will display the
158 ; error in SuperBASIC, but EXEC/EX will not.
159 ;---------------------------------------------------------------
160 errorExit
161     move.l  d0,d3               ; Error code we want to return
162
163 ;---------------------------------------------------------------
164 ; Kill myself when an error was detected, or at EOF.
165 ;---------------------------------------------------------------
166 suicide
167     moveq   #mt_frjob,d0        ; This job will die soon
168     moveq   #me,d1
169     trap    #1
```

Listing 4.9: Ql2win: Exit

There's not much to see here. We arrive at `allDone` when we hit End Of File on the input file and at `errorExit` if any errors were detected. The job then commits suicide by removing itself from the system, returning any error codes in `D3` as required. These errors will be seen only if you executed the utility with the `EXEC_W` or `EW` commands. `EXEC` and `EW` do not wait for the job to complete so cannot know in advance what, if any, errors will occur.

Finally, we have Listing 4.10, which is where we define the buffer which will be used to read data into from the source file, and write data out of to the destination file.

As previously mentioned, the buffer is two bytes larger (although it only needs one) than we tell QDOSMSQ as we need that extra one byte to insert a carriage return character, if necessary.

```
170 ;---------------------------------------------------------------
171 ; Read/write buffer. The buffer is 2 bytes longer than we need
172 ; as there needs to be room to insert the required CRLF in
173 ; place of the LF.
174 ;---------------------------------------------------------------
175 buffSize    equ     64*2        ; Buffer Size
176 buffer      ds.b    buffSize+2  ; Buffer
```

Listing 4.10: Ql2win: Buffer

4.2 Filter Chains

As mentioned already, this is a YAF. It checks that you supply exactly two channels or file names on the command line and if it doesn't find exactly two, it will exit with a bad parameter error. I was thinking "what if I wanted to check my code was working and pass the output to another filter, would that work?" I just tried it out just to see.

I was working on the assumption that Tony Tebby et al, had been smart enough[3] to ensure that chains of filter programs would be set up correctly by the `EXEC_W` or `EW` commands and things would just work. My first attempt was this:

```
EX ram1_ql2win_bin, ram1_ql_text_file TO ram1_hexdump_bin, #1
```

It did indeed work as expected, the input file, `ram1_ql_text_file`, was passed into the `ql2win` filter and had carriage returns added where necessary. The output from that filter was written directly to the input channel of the `hexdump` filter, thanks to the `TO` separator, from where, the output was displayed on screen in channel 2.

This was extremely handy for testing as I could see the carriage returns added in the correct places without having to create and open additional files.

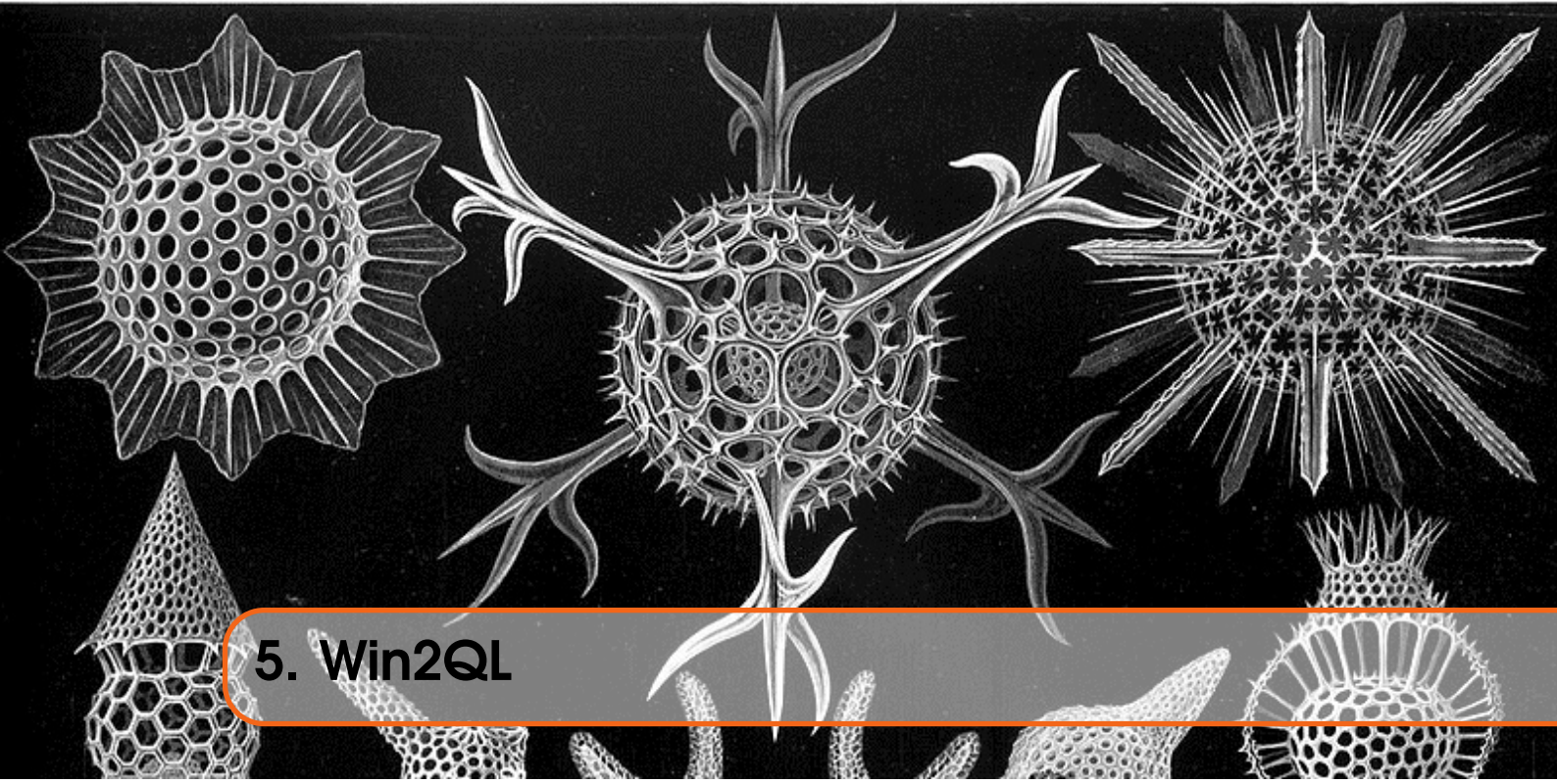
Of course, lots of YAFs can be strung together to create the final output. Here's another silly example:

```
EX ram1_ql2win_bin, ram1_ql_text_file TO ram1_win2ql_bin TO
   ⟹ ram1_hexdump_bin, #1
```

That's all one command by the way. The text file in QL format is filtered to Windows format and then passed through YAF to remove the newly added carriage returns and finally, for now, displayed on screen in hexadecimal. I used this to ensure that the output file from my two filters was identical to the text file used as the input to the test.

Once again, the `TO` separator has made sure that there are at least an input and an output channel for the filter in the chain, even though there appears to be none.

[3] And, indeed, they *were* smart enough!



5. Win2QL

So that's a utility to convert files created on the QL (or Linux!) into a format that Windows is happy with. Admittedly, even Notepad these days is able to cope with QL/Linux line endings, but it's nice to have the correct format I suppose.

Win2ql is a utility, a YAF, to convert from Windows format text files to QL format text files. It reads each line of the input file, strips off the carriage returns that it finds immediately prior to a linefeed, and writes out the adjusted buffer to the output file.

The vast majority of the code is exactly the same as discussed in the previous chapter so most of what was described there is the same and is not discussed further.

The code in the download is obviously the full utility, but for the rest of this chapter, only the changes in the file `win2ql_asm` will be discussed.

5.1 Changes From Ql2win

The first difference is in the comments at the top of the code file. Listing 4.1 in the previous chapter has been slightly amended, but only as far as the comments are concerned, none of the equates are affected. Listing 5.1 shows the new comments.

```
1  ;--------------------------------------------------------------
2  ; WIN2QL:
3  ;
4  ; This filter converts Windows line endings to QL or Linux
5  ; format.
6  ;
7  ; EX win2ql_bin, input_file, output_file_or_channel
8  ;--------------------------------------------------------------
9  ; 21/02/2021 NDunbar Created for QDOSMSQ Assembly Mailing List
10 ;--------------------------------------------------------------
11 ; (c) Norman Dunbar, 2021. Permission granted for unlimited use
```

```
12 ; or abuse, without attribution being required. Just enjoy!
13 ;-----------------------------------------------------------------------
```

Listing 5.1: Win2ql: Comments

The next change is in the job name. Listing 5.2 shows the new job header with the amended job name. The line numbers should, *hopefully*, match those in the listings being changed, those from Ql2win.

```
35 ;=======================================================================
36 ; Here begins the code.
37 ;-----------------------------------------------------------------------
38 ; Stack on entry:
39 ;
40 ; $06(a7) = Output file channel id.
41 ; $02(a7) = Source file channel id.
42 ; $00(a7) = How many channels? Should be $02.
43 ;=======================================================================
44 start
45     bra.s    checkStack
46     dc.l     $00
47     dc.w     $4afb
48 name
49     dc.w     name_end-name-2
50     dc.b     'WIN2QL'
51 name_end     equ        *
52
53 version
54     dc.w     vers_end-version-2
55     dc.b     'Version 1.00'
56 vers_end     equ        *
```

Listing 5.2: Win2ql: Job Header

There's a large gap before we hit the next change. Listing 4.7 changes to the code shown in Listing 5.3.

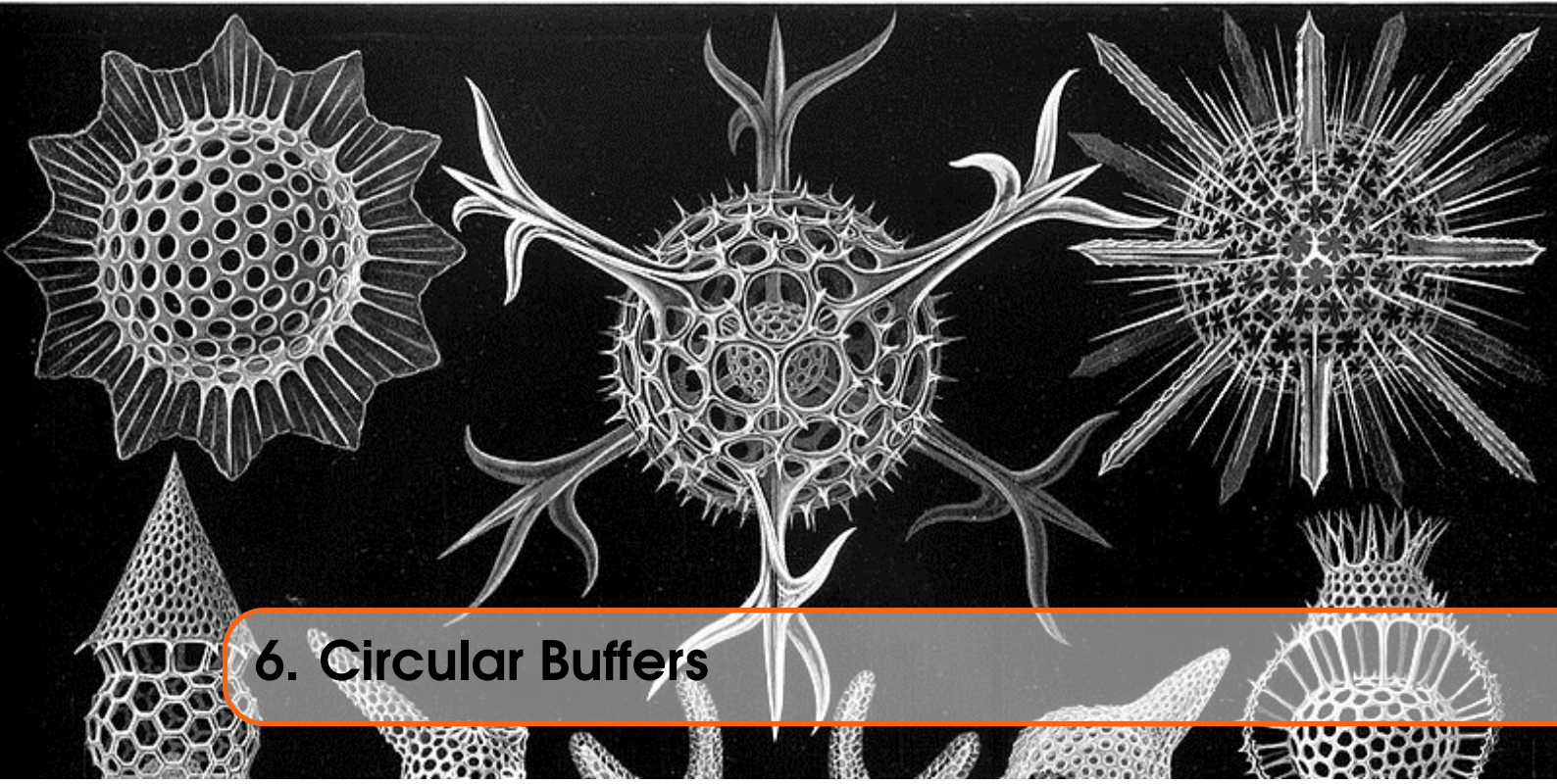
```
120 ;-----------------------------------------------------------------------
121 ; We have read at least the end of a line and have the LF at
122 ; the correct place in the buffer. If the character before the
123 ; LF is a CR remove it and write out, otherwise just write out
124 ; what we have, it's not in Windows format.
125 ;-----------------------------------------------------------------------
126     cmpi.b   #cr,-2(a1)          ; Windows format?
127     bne.s    putLine             ; No, write out what we have
128     move.b   #lf,-2(a1)          ; Replace CR with LF
129     subq.w   #1,d2               ; Update count for the missing CR
```

Listing 5.3: Win2ql: Removing a CR

As before, in Ql2win, we check if the character prior to the trailing linefeed is a carriage return. In this case, we are expecting it to be found, but if not, we can assume that this line, at least, is not in Windows format and skip off to writing the line to the output channel.

If we did find a carriage return, all we have to do is overwrite it with a linefeed and adjust the line length in `D2.W` to account for a single character less in the buffer.

The rest of the code is identical to Ql2win and has been discussed already.



6. Circular Buffers

A circular buffer is a device whereby data are written to it in order and removed from it in the same order – it's a *First In, First Out* queue, or FIFO, in other words. It is named *circular* because when data are written to the last location, the next byte stored will be at the beginning – as if the two ends of the buffer are joined together in, well, a circle! Wikipedia has a good description at https://en.wikipedia.org/wiki/Circular_buffer if you wish to read further.

I learned about them when writing my book *Arduino Software Internals*[1], while looking at the code for the Serial interface.

The application code can write data to the buffer, and as long as it doesn't get full, the USART code can be pulling data out of the same buffer and sending it out onto the serial interface pins. Is this useful for the QL I hear you think? Who knows, but I thought it would be an interesting exercise to convert from C++ to Assembly Language, just for fun[2].

The plan is, that a buffer can be allocated at a certain size, and then used and abused as required. As the code is required to use the Modulus operator to ensure that various calculations wrap around properly, the only restrictions of these buffers are:

- The size requested must be a power of two;
- The size requested must be a minimum of 2 bytes;
- The size requested must be a maximum of 32768 bytes;

Other than that, the sky is the limit!

[1] *https://www.apress.com/gb/book/9781484257890* and *https://www.amazon.co.uk/Arduino-Software-Internals-Complete-Language/dp/1484257898*

[2] For certain values of "fun" perhaps!

6.1 How Big is my Buffer?

You might think that a buffer is as big as requested? Assuming the request was for a valid power of two in size of course. A foible of circular buffers of this format is that they always end up losing a single byte of storage. Why is this?

Imagine a brand new empty buffer, let's say it's only two bytes in size[3].

As the buffer has just been allocated, $head = tail = 0$ and the buffer is officially empty. Nothing has been written to the buffer and nothing is available to be read. So far so good!

The application wants to write a byte to the buffer, so:

- Increment the head offset by 1. $Head = 1$.
- $Head\ MOD\ 2$, which is required to account for any wrap around, leaves $Head = 1$.
- Compare with the tail offset, 0. They are different, so the buffer has space available.
- Store the data byte at offset 1.
- Update the head offset in the buffer header.

This executes successfully and on return, $head = 1$ and $tail = 0$. The buffer has one byte in the data area. As you can see, the head pointer was incremented before storing the data byte and the new byte stored at that location. This means that currently, byte zero in the data area has been skipped over. Bear this in mind.

The application now wants to write a second byte to the buffer. The execution of this proceeds as follows:

- Increment the head offset by 1. $Head = 2$.
- $Head\ MOD\ 2$, leaves $Head = 0$
- Compare with the tail offset, 0. They are equal, so the buffer has no space available!
- The second byte cannot be stored in the buffer!

So, we allocated a two byte buffer by can only store a single byte in it. There are other buffer structures which store the count of data bytes written and adjust this on each read or write, these buffers have the full compliment of space available for the overhead of a bit of extra processing.

Ok, why are we comparing the incremented head offset with the unincremented tail offset? If we didn't, then the test would have to be whether the two offsets were equal, which means we can't tell if the buffer is empty or full.

6.2 Buffer Structure

The buffer is allocated as a chunk of memory where the size asked for by the user must be a power of two with a maximum size of 32,768 bytes which happens to be the largest power that fits into a word. Why a word? The buffer is made up of two parts, the data area of the requested size and a 10 byte header added to the front of the data area. The header is used to hold information about the buffer – its size plus the *head* and *tail* offsets into the data area – and these are word sized.

The 10 byte header stores the following information about the buffer:

- A buffer identifier, currently "cB60", which is used to identify that the address in `A3.L` is indeed a circular buffer.
- The data size. This is the actual size of the data area allocated for the buffer and does not include the 10 bytes added for the header. In all the code, `A3.L` points at this address, bit at

[3] Because that means I have less typing to do in explaining it!

the buffer identifier.
- The head offset. This defines the offset into the data area where the most recent byte was *written* to the buffer. The head will be adjusted to the next offset when a new byte is to be added to the buffer.
- The tail offset. This defines the offset into the data area where the most recent byte was *read* from the buffer. The tail will be adjusted to locate the next offset, when the next request for a byte is processed.

The structure of our circular buffers will be as shown in Figure 6.1.

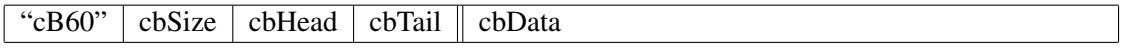


Figure 6.1: Circular Buffer structure

You can see the simplicity of the whole thing, it’s just the data area with the afore mentioned 6 byte header.

6.3 The Buffer Handling Code

The code functions we will create for this article will expect a buffer address to be passed in `A3.L` and it is required that the address pointed top by `A3.L` is that of the `cbSize` field in the buffer. However, as the initialisation of each function will retrieve the fields required from the buffer header, `A3.L` will end up pointing at the start of the data area for the body of the function. Because of this, I need to use some negative offsets when the code needs to refer to the header fields.

```
1  cbSize    equ -6      ; How big is my buffer's data area?
2  cbHead    equ -4      ; Where is my head? Last byte inserted.
3  cbTail    equ -2      ; Where is my tail? Last byte removed.
4
5  buffID    equ -4      ; Buffer identifier offset from cbSize.
6  hdrSize   equ 10      ; Size of buffer header
7
8  bufferID  equ "cB60"  ; Buffer identifier
9
10 ; Not required for GWASS/GWASL but maybe for QMAC.
11 ;MT_ALCHP equ $18     ; Allocate common heap
12 ;MT_RECHP equ $19     ; Release common heap
```

All the code for this article can be found in the code files supplied in the download. The various functions explained below are found in `cBuffer_asm`.

6.3.1 Allocate a New Buffer

Allocating a new circular buffer is a simple matter of taking the requested space, rounding it up to the next power of two, add making it a long word to fit into `D1` and adding the 6 extra bytes required for the header. A chunk of common heap is then requested from QDOSMSQ and if it was allocated, the header fields are filled in.

Listing 6.1 is the code to allocate a new circular buffer. This code calls the `checkSize` procedure to potentially adjust the buffer size to a power of two. That code can be seen in Listing 6.2.

```
1  ;-------------------------------------------------------------------------
```

```
; Allocate Buffer
;-------------------------------------------------------------------------
; Allocates memory for a new circular buffer. The size passed
; must be a power of 2 and cannot be larger than a word. This
; limits a buffer to a maximum size of 32,768 bytes. 1 of which
; will be unusable.
;
; ENTRY:
;
; D0.W = Size of buffer. Power of two, 32768 maximum.
;
; EXIT:
;
; D0.L = Error code. (From MT_ALCHP)
;      = 0 if the buffer was created.
;      <>0 if the buffer failed to create.
;
; A3.L = Buffer address. (A3 -> cbSize in the buffer.)
;
; All other registers are preserved.
;-------------------------------------------------------------------------
allocateBuffer
        movem.l d1-d3/a0-a2,-(a7) ; Save working registers
        bsr.s checkSize           ; Returns D0.L as a power of 2
        move.l d0,d1              ; D1.L = space required
        move.w d1,-(a7)           ; Save rounded requested size
        addq.l #hdrSize,d1        ; Adjust for header space
        moveq #mt_alchp,d0        ; Allocate common heap
        moveq #-1,d2              ; Current job is owner
        trap #1
        move.w (a7)+,d1           ; Restore rounded size
        tst.l d0                  ; Do we allocate some heap?
        bne.s abExit              ; No

        move.l a0,a3              ; Buffer address in A3.L
        move.l #bufferID,(a3)+    ; Buffer identifier at -4(a3)
        subq.w #hdrSize,d1        ; Size of buffer data area
        move.w d1,(a3)            ; Set cbSize
        clr.l 2(a3)               ; Set cbHead = cbTail = 0

abExit
        movem.l (a7)+,d1-d3/a0-a2 ; Restore working registers
        rts
```

Listing 6.1: Allocating a circular buffer

All registers except D0 and A3 are preserved by the allocate Buffer routine. The code is called with the required buffer size in D0.W and if successful, D0 will hold zero and A3.L will return the buffer address. In the case of an error, D0 will return the error code from MT_ALCHP and A3 will be unchanged.

Register D1.W is stacked before being adjusted for the header size, and is restored after the trap. The trap call to allocate common heap returns the number of bytes allocated in D1.L, however, this is not necessarily the same as the number of bytes requested as the requested size will be rounded up by QDOSMSQ.

When testing, I asked for an 8 byte buffer which works out at 18 bytes with the header included, I received a chunk of common heap which was 48 bytes in size. That messed up the `cbSize` field in the header and was an interesting bug to track down! By saving `D1.W` I can set the header to the correct buffer size.

6.3.2 Buffer Size Adjustments

In order to protect the programmer from him or herself, the allocation of a buffer will check that the size requested is within range. The code in the `checkSize` routine takes the value in `D0.W` and rounds it up to the next power of two, unless its already a power. The resulting value, in `D0.L` is then adjusted to a maximum of 32768 or a minimum of 8 bytes.

The code should be reasonably familiar as it was in Issue 8 of this somewhat irregular eMagazine. Listing 6.2 shows the code to check and make the adjustments as necessary.

```
1  ;-------------------------------------------------------------------------
2  ; Check Size
3  ;-------------------------------------------------------------------------
4  ; Check the requested buffer size in D0 and round it up to the
5  ; next largest power of two if not already a power. If less
6  ; than 8 bytes, make it 8. If more than 32768, then make it
7  ; 32768.
8  ;
9  ; ENTRY:
10 ;
11 ; D0.W = Size of buffer.
12 ;
13 ; EXIT:
14 ;
15 ; D0.L = Potentially adjusted buffer size.
16 ; D1.L = D0.L.
17 ;
18 ; All other registers are preserved.
19 ;-------------------------------------------------------------------------
20 ; Algorithm:
21 ;
22 ; Value = Value - 1
23 ; REPEAT LOOP
24 ;    Temp = Value & (Value - 1)
25 ;    If Temp = 0, return min(max(2*Value, 8), 32768)
26 ;    Value = Temp
27 ; END LOOP
28 ;
29 ; D1.L = Value
30 ; D0.L = Temp
31 ;-------------------------------------------------------------------------
32 checkSize
33         moveq #0,d1
34         move.w d0,d1          ; D1.L = Value
35
36         subq.l #1,d1          ; In case Value is a power already
37
38 csLoop
39         move.l d1,d0          ; Temp = Value
40         subq.l #1,d0          ; Temp = (Value - 1)
```

```
41          and.l d1,d0             ; Temp = Value & (Value - 1)
42          beq.s csRange           ; If Temp = Zero = no more set bits
43          move.l d0,d1            ; Value = Temp
44          bne.s csLoop            ; Keep going
45
46  csRange
47          lsl.l #1,d1             ; Value = Value * 2
48          move.l d1,d0            ; To return the new value
49
50  csMin
51          cmpi.l #7,d0            ; Minimum is 8
52          bhi.s csMax             ; Bigger than 7 is ok
53          moveq #8,d0             ; Result is 8
54          bra.s csExit            ; Done
55
56  csMax
57          cmpi.l #$8000,d0        ; Maximum is 32768
58          ble.s csExit            ; Equal/Smaller than 32768
59          move.l #$8000,d0        ; Result is 32768
60
61  csExit
62          rts
```

Listing 6.2: Adjusting a buffer's size

In order to test if a number is a power of two, start with one less than the number – in case it's already a power – then repeatedly assign the value with $(value\ AND\ (value-1))$ and when the result is zero, you have the power of two *below* the original number, to get the next one, multiply by two.

The result, in `D0.L`, is now a power of two. Given that a buffer size of less than 8 is most likely a waste of time, the buffer size is rounded up to 8 if smaller. If the size exceeds the maximum that a word can hold, 32768, it is adjusted down to 32768.

6.3.3 Free a Buffer

Listing 6.3 shows the code to free a circular buffer after it is no longer required. Obviously in a job, this is not strictly necessary as QDOSMSQ will tidy up the jobs allocated heaps space on exit, however, it's best to be neat and tidy, plus, deallocating the space when no longer needed can free the space for other tasks to utilise.

```
1  ;--------------------------------------------------------------------
2  ; Free Buffer
3  ;--------------------------------------------------------------------
4  ; Deallocates memory for a circular buffer. The buffer Id is
5  ; cleared to hopefully prevent deleted buffers from being used.
6  ;
7  ; ENTRY:
8  ;
9  ; A3.L = Buffer address. (A3 = cbSize in the buffer.)
10 ;
11 ; EXIT:
12 ;
13 ; D0.L = Error code.
14 ;      = 0 if the byte was added to the buffer.
15 ;      = 1 if the buffer is full, so D1 was not added.
```

```
;       = 2 if the buffer address was invalid.
;
; A3.L = 0. Buffer now invalid.
;
; All other registers are preserved.
;-------------------------------------------------------------------------
freeBuffer
        bsr.s bufferCheck       ; Won't return unless valid
        movem.l d0-d3/a0-a2,-(a7) ; Save working registers
        move.l #0,buffID(a3)    ; Delete buffer id
        move.l a3,a0            ; Buffer address to reclaim
        moveq #MT_RECHP,d0      ; Release common heap
        trap #1

fbExit
        movem.l (a7)+,d0-d3/a0-a2 ; Restore working registers
        move.l #0,a3            ; Buffer deleted.
        rts
```

Listing 6.3: Freeing a circular buffer

There's not a lot to explain here, the base address of the buffer is passed in `A3` as usual, and is simply copied to `A0` in order for the trap to release common heap space, to work. releasing a heap never fails with any errors, so none are checked for.

All registers, except `A3`, are preserved by this code.

6.3.4 Buffer Check

Before operating on a buffer, it's wise to attempt a bit of validation to try and prevent things going awry when the code starts accessing areas of RAM which are not actually circular buffers!

The address of a buffer, as previously discussed, points at the size word in the header, however, just before the size word is an identifier which is hard coded to be the text "cB60" – for no real reason – and the `bufferCheck` procedure, shown in Listing 6.4, checks that this long word does exist in the correct place.

If, for some unknown reason, the identifier is not found, then something has gone wrong. `D0.L` is set to 2 to indicate an invalid buffer, and the callers return address is removed from the stack allowing the code to return to the caller's caller with the error code.

```
;-------------------------------------------------------------------------
; Buffer Check
;-------------------------------------------------------------------------
; A buffer address in A3 is checked for an attempt at validity
; in that an address of zero is considered invalid, whereas any
; other value is possibly valid! Hard to determine, I know.
;
; Any function that manipulates buffers should (!) BSR to here
; before doing any register saving etc. Else, carnage will be
; the result!
;
; NOTE: The return is to the previous caller on error. This
;       code will only return to the caller if the buffer is
;       non-zero. So:
;
```

```
16 ;        codeXxx calls addByte, for example.
17 ;        addByte calls here to check buffer.
18 ;        If A3 is zero, return to codeXxx with D0 = 2.
19 ;        Else, return to addByte to add a byte.
20 ;
21 ; ENTRY:
22 ;
23 ; A3.L = Buffer address. (A3 = cbSize in the buffer.)
24 ;
25 ; EXIT:
26 ;
27 ; D0.L = Error code.
28 ;      = 2 The buffer is invalid.
29 ;-------------------------------------------------------------------------
30 bufferCheck
31         cmpi.w #bufferID,-4(a3) ; Is this a buffer?
32         beq.s bcExit            ; Buffer ok, return to caller
33         moveq.l #2,d0           ; Buffer is bad
34         addq.l #4,a7            ; Caller address ignored
35
36 bcExit
37         rts
```

Listing 6.4: Validating a buffer address

6.3.5 Write Data to a Buffer

When adding a byte to a buffer, the buffer cannot be full up, that indicates an error condition. If the buffer has free space, then the head offset is adjusted to the next free space and the data byte stored at that offset into the data area of the buffer.

Listing 6.5 shows the code for the `addByte` procedure..

```
1  ;-------------------------------------------------------------------------
2  ; Add Byte
3  ;-------------------------------------------------------------------------
4  ; Adds one byte to a circular buffer.
5  ;
6  ; ENTRY:
7  ;
8  ; D1.B = Byte to be added.
9  ; A3.L = Buffer address. (A3 = cbSize in the buffer.)
10 ;
11 ; EXIT:
12 ;
13 ; D0.L = Error code.
14 ;      = 0 if the byte was added to the buffer.
15 ;      = 1 if the buffer is full, so D1 was not added.
16 ;      = 2 if the buffer address was invalid.
17 ;
18 ; All other registers are preserved.
19 ;-------------------------------------------------------------------------
20 addByte
21         bsr.s bufferCheck       ; Won't return unless valid
22         movem.l d1/d4-d5/a3,-(a7) ; Save working registers
23
```

```
abIsFull
        bsr.s isFull            ; Preserves all registers
        beq.s abFullUp          ; Yes, bale out

        move.w (a3)+,d5         ; Size of buffer
        subq.w #1,d5            ; We need one less than size
        move.w (a3)+,d4         ; Where is the head?
        addq.l #2,a3            ; Skip over the tail, not used
        addq.w #1,d4            ; New head pointer
        and.w d5,d4             ; Wrap around if necessary
        move.b d1,(a3,d4.w)     ; Store new byte
        move.w d4,cbHead(a3)    ; New head saved
        moveq #0,d0             ; One byte added
        bra.s abExit            ; Done, no errors.

abFullUp
        moveq #1,d0             ; Buffer full, can't add D1

abExit
        movem.l (a7)+,d1/d4-d5/a3 ; Restore working registers
        rts
```

Listing 6.5: Writing data to a buffer

The code starts by calling the `isFull` routine to determine if the buffer is full. If `D0` is returned as zero, then the buffer is indeed full and we bale out with an error code in `D0`. We cannot add this byte to the buffer.

`D5` is then set to the buffer size minus 1. We use this to MOD the head offset when it is incremented, to enable it to wrap around from the final byte to the start byte, if required. The head offset is copied into `D4` and as we don't need the tail, `A3` is incremented past it to point at the start of the data area of the buffer. The data byte in D1 is then stored in the buffer at the new head position.

The new head offset is written back to the correct location in the buffer header, which is at a negative offset from the current value in `A3`.

The sharp eyed and quick brained amongst you may be thinking, "*Hmmmm, what about that* move.b d1,(a3,d4.w) *instruction, to store the new data byte, as the index register is a word, surely it will be sign extended? The maximum buffer is 32768 after all and that's got a sign bit of 1 after all.*"

Funnily enough, I thought that too for a bit, however, the absolute maximum value that the head or tail offsets can take is one less than the buffer size – buffer offsets are from zero – so that makes 32767 or $7FFF the maximum, and that doesn't have a leading 1 in the sign bit, so all is well.

6.3.6 Read Data from a Buffer

Reading data from a buffer is quite simple. If the buffer is empty, then there's nothing to do as there's nothing to actually read. Assuming the buffer does have data, then the tail offset is incremented, wrapping around if necessary, and the byte at that offset is obtained. The new tail pointer is then stored in the buffer header.

Listing 6.6 shows the `getByte` code.

```
;-------------------------------------------------------------------------
; Get Byte
```

```
3  ;-------------------------------------------------------------------
4  ; Gets one byte from a circular buffer.
5  ;
6  ; ENTRY:
7  ;
8  ; A3.L = Buffer address. (A3 = cbSize in the buffer.)
9  ;
10 ; EXIT:
11 ;
12 ; D0.L = Error code.
13 ;      = 0 if the byte was retrieved from the buffer.
14 ;      = 1 if the buffer is empty.
15 ;      = 2 if the buffer address was invalid.
16 ;
17 ; D1.B = The retrieved byte, if the buffer was not empty.
18 ;      = Preserved if the buffer was empty.
19 ;
20 ; All other registers are preserved.
21 ;-------------------------------------------------------------------
22 getByte
23         bsr.s bufferCheck        ; Won't return unless valid
24         movem.l d4-d6/a3,-(a7)   ; Save working registers
25
26 gbIsEmpty
27         bsr.s isEmpty            ; Is the buffer empty?
28         beq.s gbEmpty            ; Yes, bale out
29
30         move.w (a3)+,d5          ; Size of buffer
31         subq.w #1,d5             ; We need one less than size
32         addq.l #2,a3             ; Skip over the head
33         move.w (a3)+,d6          ; Get the tail?
34         addq.w #1,d6             ; New tail pointer
35         and.w d5,d6              ; Wrap around if necessary
36         move.b (a3,d6.w),d1      ; Fetch byte
37         move.w d6,cbTail(a3)     ; New tail saved
38         moveq #0,d0              ; One byte retrieved
39         bra.s gbExit             ; Done, no errors
40
41 gbEmpty
42         moveq #1,d0              ; Buffer empty, can't retrieve
43
44 gbExit
45         movem.l (a7)+,d4-d6/a3   ; Restore working registers
46         rts
```

Listing 6.6: Reading data from a buffer

The code begins by calling out to `isEmpty` and if the buffer is empty, bales out setting `D0` to 1 to show a read from an empty buffer was attempted.

The buffer size is copied into `D5.W` and decremented. This is used later to MOD the new tail offset to make sure it wraps around, if required. The head offset is not required so is skipped over leaving `A3.L` pointing at the tail, which is read into `D6`.W. The tail offset is then incremented, wrapping as appropriate to give the offset into the data area that we will read from.

The data byte at the calculated offset is copied into D1 and the new tail offset stored in the buffer's

header. D0 is cleared to show that no errors occurred.

6.3.7 Is the Buffer Full?

The buffer is full up whenever the tail offset is 1 byte larger than the head. The next data byte to be written to the buffer will be stored at:

$$head + 1 \; MOD \; buffer\_size$$

The tail offset, whatever it currently happens to be, is where the most recent byte was read from the buffer. As previously explained, we have to compare the *incremented head* offset with the *current tail* offset or we will be unable to determine if the buffer is empty or full when both are equal.

It is this increment to the head offset which causes the loss of a single byte of storage in the buffer.

```
1  ;-------------------------------------------------------------------
2  ; Is Full?
3  ;-------------------------------------------------------------------
4  ; Checks if the buffer passed in A3 is full. A buffer is full
5  ; when (Head + 1) == Tail.
6  ;
7  ; ENTRY:
8  ;
9  ; A3.L = Buffer address. (A3 = cbSize in the buffer.)
10 ;
11 ; EXIT:
12 ;
13 ; D0.L = Return code.
14 ;      = 0 if the buffer is full. Z set.
15 ;      = 1 if the buffer is not full. Z clear.
16 ;      = 2 if the buffer address was invalid.
17 ;
18 ; All other registers are preserved.
19 ;-------------------------------------------------------------------
20 isFull
21         bsr bufferCheck          ; Won't return unless valid
22         movem.l d4-d5/a3,-(a7)   ; Save the workers
23         move.w (a3)+,d4          ; Get the buffer size
24         subq.w #1,d4             ; Minus 1 for MOD
25         move.w (a3)+,d5          ; Get the head offset
26         addq.w #1,d5             ; Next head offset
27         and.w d4,d5              ; MOD buffer size
28         cmp.w (a3),d5            ; Same as tail?
29         beq.s ifFull             ; Buffer is full
30         moveq #1,d0              ; Not full
31         bra.s ifExit             ; Bale out
32
33 ifFull
34         moveq #0,d0              ; Buffer is full
35
36 ifExit
37         movem.l (a7)+,d4-d5/a3   ; Restore the workers
38         rts
```

Listing 6.7: Is buffer full

The code is simple enough, the buffer's size is copied into `D4.W` and decremented ready for the MOD to take place. The head offset is copied into `D5.W` and incremented. The new value is ANDed with `D4.W` to cope with t he new value needing to wrap around to the beginning of the buffer. The new value is compared with the tail offset and if they are equal, the buffer is full. This is indicated by a return value of zero in `D0`. Returning 1 in `D0` indicates that the buffer is not full.

6.3.8 Is the Buffer Empty?

This is rather easy. If the head pointer equals the tail pointer, then the buffer is currently empty. For a brand new buffer, both offsets are zero and the buffer is definitely empty.

If, on the other hand, 10 bytes had been added since new, and none read back yet, the head will be 10 while the tail will be still zero. Don't I mean 9 for the head? No, definitely 10 because while the head (and tail) starts at zero in a new buffer, it is incremented by 1 *before* storing a new byte as explained above. The first byte added will be at offset 1 for a new buffer, the second at offset 2 and so the tenth will be at offset 10.

After reading back the 10 bytes, the tail offset will also be at 10, so both are equal and the buffer is indeed empty.

Interestingly, a new, empty, buffer need not have the head and tail offsets set to zero, it makes no difference where they both point, provided they point at the same offset.

```
1  ;-------------------------------------------------------------------------
2  ; Is Empty?
3  ;-------------------------------------------------------------------------
4  ; Checks if the buffer passed in A3 is empty. A buffer is empty
5  ; when Head == Tail.
6  ;
7  ; ENTRY:
8  ;
9  ; A3.L = Buffer address. (A3 = cbSize in the buffer.)
10 ;
11 ; EXIT:
12 ;
13 ; D0.L = Return code.
14 ;      = 0 if the buffer is empty. Z set.
15 ;      = 1 if the buffer is not empty. Z clear.
16 ;      = 2 if the buffer address was invalid.
17 ;
18 ; All other registers are preserved.
19 ;-------------------------------------------------------------------------
20 isEmpty
21         bsr bufferCheck          ; Won't return unless valid
22         move.w 2(a3),d0          ; Head offset
23         cmp.w 4(a3),d0           ; Head = Tail?
24         beq.s ieEmpty            ; Yes
25         moveq #1,d0              ; Not empty
26         rts
27
28 ieEmpty
29         moveq #0,d0              ; No
30         rts
```

Listing 6.8: Is buffer empty

6.3.9 How Much Space is Used?

The used space in a buffer is calculated as:

$$(buffer\_size + head - tail)\ MOD\ buffer\_size$$

If we assume a buffer state as shown in Figure 6.2 we can see a simple example where 4 bytes of data have been written to a new buffer but nothing has been read back yet. The first byte has an unknown value as it was never written to since the buffer was created. The 4 bytes written are 'A', 'B', 'C' and 'D'.

cbSize	cbHead	cbTail	cbData							
8	4	0	$?_{tail}$	A	B	C	D_{head}	?	?	?

Figure 6.2: Circular Buffer space used - simple example

This is an obvious one, there are 4 bytes used, we can see that plainly. And the calculation gives the correct result:

$$\begin{aligned}(buffer\_size + head - tail)\ &MOD\ buffer\_size\\ (8+4-0)\ &MOD\ 8\\ 12\ &MOD\ 8\end{aligned}$$

And 12 *MOD* 8 is indeed 4.

How about when the buffer has been used for a while and the buffer state resembles Figure 6.3.

cbSize	cbHead	cbTail	cbData							
8	2	5	Y	Z	A_{head}	C	D	E_{tail}	F	G

Figure 6.3: Circular Buffer space used - complex example

In this example, the last byte written was at offset 2 ($head = 2$), the 'A', while the last byte read back was at offset 5 (tail = 5), the 'E'. We can plainly see that the bytes F, G, Y, Z and A have yet to be read and so must be included in the count, while the data bytes C, D and E have already been read back and are thus classed as free space now.

$$\begin{aligned}(buffer\_size + head - tail)\ &MOD\ buffer\_size\\ (8+2-5)\ &MOD\ 8\\ 5\ &MOD\ 8\end{aligned}$$

And 5 *MOD* 8 is 5 and there are 5 unread bytes in the buffer – F, G, Y, Z and A.

Listing 6.9 shows the code to work out how much data is available in a buffer.

```
1 ;------------------------------------------------------------------------
2 ; Get Used
3 ;------------------------------------------------------------------------
```

```
4  ; Returns the space used in a buffer. This is (size + head -
5  ; tail) MOD size.
6  ;
7  ; ENTRY:
8  ;
9  ; A3.L = Buffer address. (A3 = cbSize in the buffer.)
10 ;
11 ; EXIT:
12 ;
13 ; D0.W = Used space.
14 ;      = 0 if the buffer is empty. Z set.
15 ;      = 2 if the buffer address was invalid.
16 ;      <> 0 = used space. Z clear.
17 ;
18 ; All other registers are preserved.
19 ;-------------------------------------------------------------------
20 getUsed
21         bsr bufferCheck          ; Won't return unless valid
22         movem.l d5/a3,-(a7)      ; Save the workers
23
24 guIsEmpty
25         bsr.s isEmpty            ; Is buffer empty?
26         beq.s guExit             ; Yes, D0 = 0, bale out
27
28         move.w (a3)+,d0          ; Buffer size
29         move.w d0,d5             ; Buffer size again
30         subq.w #1,d5             ; For MOD
31         add.w (a3)+,d0           ; Add on head
32         sub.w (a3),d0            ; Minus tail
33         andw.l d5,d0             ; MOD size
34
35 guExit
36         movem.l (a7)+,d5/a3      ; Restore the workers
37         rts
```

Listing 6.9: Buffer space used

If the buffer is empty, then we exit from the code with `D0` holding zero. The buffer size is copied into `D0.W` and `D5.W`. `D5.W` is then decremented ready for the MOD operation later. The current head offset is added to `D0.W` and the tail offset is subtracted. `D0.W` is finally ANDed with `D5.W` to obtain the final result for $(buffer\_size + head - tail)\ MOD\ buffer\_size$.

Can the results ever overflow a word sized register? No. The biggest buffer allowed is 32768 which is $8000, in that buffer the maximum head offset to add on is 32767 or $7FFF, this gives 65535 or $FFFF – so it all fits into a word before subtracting the tail offset, the smallest of which is zero. The result must always fit into a word sized register, so we are good.

6.3.10 How Much Space is Free?

The space currently available in a buffer is calculated as:

$$buffer\_size - 1 - space\_used$$

Alternatively, substituting the formula for space used:

$$buffer\_size - 1 - ((buffer\_size + head - tail)\ MOD\ buffer\_size)$$

The code below in Listing6.10 does exactly this using the former formula but it could be rewritten to use the full formula of course, but when half the work has been done already, it's a shame to repeat it!

```
;-------------------------------------------------------------------
; Get Free
;-------------------------------------------------------------------
; Returns the space free in a buffer. This is size - 1 - used.
;
; ENTRY:
;
; A3.L = Buffer address. (A3 = cbSize in the buffer.)
;
; EXIT:
;
; D0.L = Space used in the buffer.
;      = 0 if the buffer is full. Z set also.
;      = 2 if the buffer address was invalid.
;
; All other registers are preserved.
;-------------------------------------------------------------------
getFree
        bsr bufferCheck         ; Won't return unless valid
        move.l a3,-(a7)         ; Save the worker

gfIsEmpty
        bsr.s getUsed           ; D0.W = used space
        neg.w d0                ; negative used size
        add.w (a3),d0           ; Add buffer size
        subq.w #1,d0            ; Minus the unusable byte

gfExit
        move.l (a7)+,a3         ; Restore the worker
        rts
```

Listing 6.10: Buffer free space

The code is quite simple here as well. `D0.W` is set to the amount of space used in the buffer, this is then negated and the buffer size is added on. The single unusable byte is then subtracted to get the final result.

6.3.11 Flushing Buffers

Listing 6.11 shows the code to flush, or empty, a buffer if this is required for any reason. The code is very simple, it sets the head and tail offsets to zero in the buffer header thus quickly emptying the buffer.

```
;-------------------------------------------------------------------
; Flush Buffer
;-------------------------------------------------------------------
; Flushes all data from the buffer. Doesn't overwrite it, only
; set the Head = Tail = 0. The size remains the same.
```

```
6  ;
7  ; ENTRY:
8  ;
9  ; A3.L = Buffer to flush. (A3 = cbSize in the buffer.)
10 ;
11 ; EXIT:
12 ;
13 ; D0.L = Error Code
14 ;      = 2 if the buffer address was invalid.
15 ; None.
16 ;
17 ; All registers are preserved.
18 ;-------------------------------------------------------------------
19 flushBuffer
20         bsr bufferCheck          ; Won't return unless valid
21         move.w #0,-cbHead(a3)    ; -cbHead is actually cbTail!
22         move.w #0,-cbTail(a3)    ; -cbTail is actually cbHead!
23         rts
```

Listing 6.11: Flushing buffers

6.3.12 Incrementing Head and Tail Offsets

The various procedures explained above all increment their offsets for head and tail, on the fly, rather than calling out to another subroutine. This is mainly because the increment code is pretty small and it's hardly worth calling a sub-routine to do the work.

Incrementing an offset is a simple case of:

$$(offset + 1)\ MOD\ buffer\_size$$

We need the MOD function as the offset has to wrap back to zero when we attempt to increment past the end of the buffer.

This is another reason why the buffer size is required to be a power of two. When those values are used, we can quickly MOD a value by ANDing with the buffer size minus 1 – as the code has been doing throughout.

For example, if our buffer size is 8 bytes, then ANDing with 7 is effectively the MOD 8 operation that we need. The offset into the buffer, head or tail, will always be from 0 to 7 inclusive. When we increment from 7 to 8, we go outside the bounds of the buffer's data area so we need to wrap back to the start.

In binary 8 is 0000 1000 and 7 is 0000 0111. If we AND them together, we get 0000 0000 which is exactly where we want to be, back at offset zero. Listing 6.12 shows an example of this in code form, where we pick up the head pointer and increment it.

```
1  incHead
2          ...
3          move.w (a3)+,d4          ; Get the buffer size
4          subq.w #1,d4             ; Minus 1 for MOD
5          move.w (a3)+,d5          ; Get the head offset
6          addq.w #1,d5             ; Next head offset
7          and.w d4,d5              ; MOD buffer size
```

```
8           ...
```

Listing 6.12: Incrementing an offset using AND

If the buffer sizes could be any value, then we would be into the realms of having to divide and take the remainder. Not that this is a huge problem, in fact, if you wish, you can rewrite the code to allow for buffer sizes which are not powers of two, call it homework!

Listing 6.13 shows an example of incrementing the head offset using the `DIVU` instruction. This requires that a *long* value in the destination register be divided by a *word* value in the source register. The high word of the destination register will contain the remainder, which is what we are after.

```
1 incHead
2           ...
3           move.w (a3)+,d4          ; Get the buffer size
4           move.w (a3)+,d5          ; Get the head offset
5           ext.l d5                 ; For DIVU
6           addq.l #1,d5             ; Next head offset
7           divu d4,d5               ; Divide by buffer size
8           swap d5                  ; Remainder in low word
9           ...
```

Listing 6.13: Incrementing an offset using DIVU

We have to take care here not to increment the head pointer until *after* it has been sign extended to a long – if the value happened to be 32767, $7FFF, and it was incremented to $8000, it would be sign extended to $FFFF 8000 and after the division, the new head pointer would be wrong. Listing 6.14 shows the code again, but this time, with the register values displayed as comments.

```
1 incHead
2           ...
3           move.w (a3)+,d4          ; D4 = xxxx 8000
4           move.w (a3)+,d5          ; D5 = xxxx 7FFF
5           ext.l d5                 ; D5 = 0000 7FFF
6           addq.l #1,d5             ; D5 = 0000 8000
7           divu d4,d5               ; D5 = 0000 0001
8           swap d5                  ; Remainder = 0000
9           ...
```

Listing 6.14: Correct offset incrementing with DIVU

And Listing 6.15 is the code where we increment the offset *before* we extend the register to a long value.

```
1 incHead
2           ...
3           move.w (a3)+,d4          ; D4 = xxxx 8000
4           move.w (a3)+,d5          ; D5 = xxxx 7FFF
5           addq.w #1,d5             ; D5 = 0000 8000
6           ext.l d5                 ; D5 = FFFF 8000
7           divu d4,d5               ; D5 = FFFF 8000
8           swap d5                  ; Remainder = FFFF !!!!!!!
9           ...
```

Listing 6.15: Incorrect offset incrementing with DIVU

We expected the remainder to be zero, and yet it actually appears to be 65535. This is, as noted, *slightly* incorrect. Read on.

6.3.13 QPC2 Bug? My Mistake?

In Listing 6.14 I show the result of the DIVU instruction as D5 = $0001 FFFF which is correct. When I was making sure that I wasn't talking rubbish again[4] regarding Listing 6.15, I traced the code through with QMON2. I could see that the DIVU instruction gave the result as D5 = $FFFF 8000 which is completely wrong!

I mentioned this problem on the this topic on QLForum[5] as I thought I *might* have found a corner case bug in QPC2. I hadn't of course.

The problem was, when I traced the code, and saw the result of the division being so obviously wrong, I leapt to the conclusion that it had to be a bug. Unfortunately, what I had neglected to do was *look at the flags*. Had I done so, I would have noticed the V flag, overflow, was set after the division. Duh!

The error of my ways was pointed out by Marcel and Tobias. Thanks to them, for being gentle with me!

The manual for the 68008 says that "*Overflow may be detected and set before the instruction completes. If the instruction detects an overflow, it sets the overflow condition code and the operands are unaffected.*"

Look at the values in the D5.L register just before and after the DIVU instruction? They are exactly the same.

Had the division been a valid one, it would have resulted in a quotient of $0001 FFFF and no remainder. The quotient is larger than a word, so wouldn't have fitted in D5.W where it should go. The overflow was detected and dealt with *exactly* as specified in the manual. And I missed it, completely! Even though I used a couple of hex calculators to check what the answer *should* have been and *knew* something was wrong with $0001 FFFF, I didn't twig the the obvious fact that the quotient was *bigger than a word*. Not that it wasn't staring me in the face!

So, don't be like me, watch the flags when something goes weird on you, and make sure you haven't done, or missed, anything silly!

NOTE: As I'm running QPC2, I have the benefit of the 68020 CPU rather than the 68008. The 68020 has a DIVU.L <ea>,Dq:Dr instruction which takes a 32 bit value in the effective address, divides by the Dq register and places a 32 bit quotient in Dq with the remainder in Dr. This will not work on a bare bones QL of course.

6.4 Test Harness

So, that's the circular buffer code written. Does it work[6]? Listing 6.16 is a small test harness to exercise the buffer handling code.

The code begins by asking for a 5 byte buffer. This is not a power of two, so it will be rounded up to 8 bytes, 7 of which can be used. The buffer will then be filled up with data, the characters 'A' through 'G' and tested to ensure it is indeed full.

After this, the data will be read back, one byte at a time until there is no more data whereupon it will be checked for emptiness and then deleted.

[4] It happens

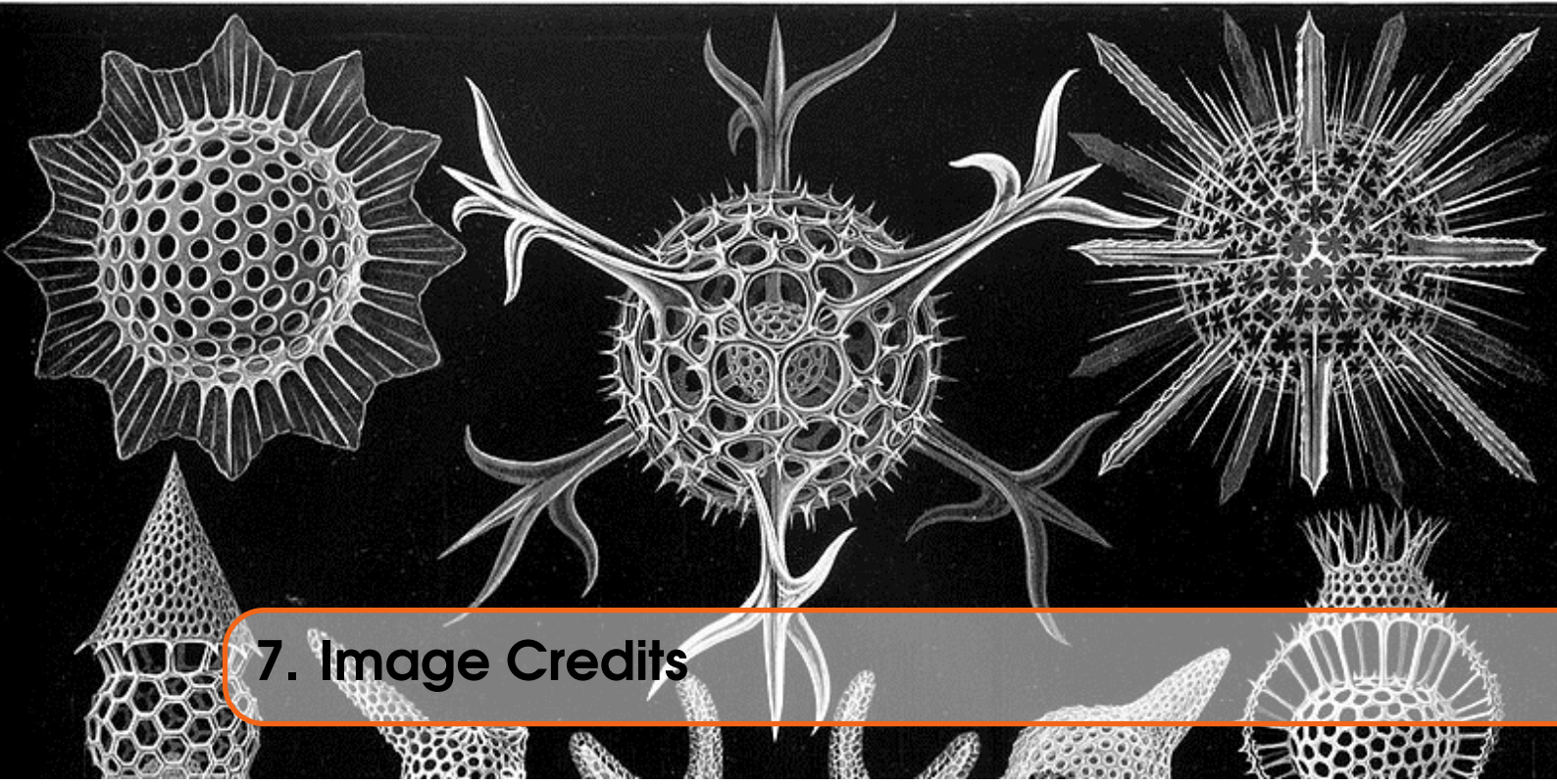
[5] The URL is https://qlforum.co.uk/viewtopic.php?f=19&t=3966&p=44307#p44295 if you have printed out a copy of this issue of the eMagazine.

[6] Given who wrote it, probably not!

```
;-------------------------------------------------------------------
; Test Harness
;-------------------------------------------------------------------
; A quick and dirty test of the cBuffers_asm code. It will:
;
; 1. Request a buffer of 5 bytes, but will get one of 8.
; 2. Write ABCDEFG to it, filling it up.
; 3. Test if it is full, D0 = 0 means it is.
; 4. Read back all the data, ABCDEFG, emptying the buffer.
; 5. Test if it is empty, D0 = 0 if so.
; 6. Free the buffer.
;
; The code here was simply traced through QMON2 to be sure that
; everything was working. It was. Error checking is few and far
; between due to the use of QMON2.
;
; Feel free to use this as a starter if you ever need to use
; circular buffers (FIFO) in your code.
;-------------------------------------------------------------------
start
        moveq #5,d0             ; Will round up to 8
        bsr allocateBuffer      ; Create buffer

        moveq #'A',d1           ; First byte
addLoop
        bsr addByte             ; Add 1 byte
        bne.s addEnd            ; Buffer full?
        addq #1,d1              ; No, next byte
        bra.s addLoop           ; And again

addEnd
        bsr isFull              ; D0 = 0 then full

getLoop
        bsr getByte             ; Get 1 byte
        bne.s getEnd            ; Buffer empty?
        bra.s getLoop           ; No, next byte

getEnd
        bsr isEmpty             ; D0 = 0 then empty
        bsr freeBuffer          ; Delete buffer

        clr.l d0                ; For SuperBASIC
        rts

        ; Pull in the cBuffer_asm code.
        in "ram1_cBuffers_asm"
```

Listing 6.16: Test harness for cBuffer code

Obviously, error checking is not a major priority here as the code was only ever intended to be used within QMON2 so that return values and such like could be checked and the buffer displayed on screen as required. Hopefully, it gives you an idea in how to use the buffer handling code. I look forward to hearing all about the programs you have written that use it.



7. Image Credits

The front cover image on this ePeriodical is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Polycystines* which are a specific kind of micro-fossil.

I have also cropped the image for use on each chapter heading page.

You can read about Polycystines on Wikipedia and there is a brief overview of the above book, also on Wikipedia, which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Polycystines have absolutely nothing to do with the QL or computing in general - in fact, I suspect they died out before electricity was invented - but I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

Not that I am suggesting, *in any way whatsoever*, that we QL fans are ancient.